第 24 章 美丽的并发

Simon Peyton Jones

免费午餐已经结束[1]。以往我们习惯于通过购买新一代 CPU 来加快程序的运行速度，但现在，这样的好时光已经一去不复返了。虽说下一代芯片会带有多个 CPU，但每个单独的 CPU 的速度却不会再变快了。所以，如果想让程序跑得更快的话，你就必须得学会编写并行程序[2]。

[1]Herb Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," Dr. Dobb's Journal, March 2005.

[2]Herb Sutter and James Larus, "Software and the concurrency revolution," ACM Queue, Vol. 3, No. 7, September 2005.

并行程序的执行是非确定性的，因而测试起来自然也就不容易；并发程序中有的 bug 甚至可能会无法重现。我对漂亮程序的定义是“简单而优雅，乃至于显然没有任何错误”，而不仅仅是“几乎没有任何明显的错误”[3]。要想编写出能够可靠运行的并行程序，程序的美感是尤其要注意的方面。可惜一般来说并行程序总归没有它们相应的非并行（顺序式的）版本漂亮；尤其是在模块性方面：并行程序的模块性相对较弱。这一点我们后面会看到。

[3]This turn of phrase is due to Tony Hoare.

本章将介绍软件事务内存（STM）。软件事务内存是一项很有前景的技术，它提出了一种针对共享内存并行处理器编程的新手段，正如刚才所言，传统并行程序的模块性较弱，而这正是软件事务内存的强项。我希望你读完本章后能和我一样对这项新技术感到振奋。当然，软件事务内存也并非万灵药，但它的确对并发领域中令人望而却步的问题发起了一次漂亮且令人鼓舞的冲击。

24.1 一个简单的例子：银行账户

假设有这样一个简单的编程任务：

编写一段程序，将钱从一个银行账户转移到另一个账户。为了简化起见，两个账户都是存放在内存里面的——也就是说你不用考虑跟数据库的交互。要求是你的代码在并发环境中也必须能够正确执行，这里所谓的并发环境也就是指可能存在多个线程同时调用你的转账函数，而所谓能够正确执行则是指任何线程都不能“看到”系统处于不一致的状态（比如看到其中一个账户显示已被取出了一笔钱然而同时另一个账户却显示还没有收到这笔钱）。

这个例子虽说有点人为捏造的味道，但它的优点是简单，因而我们便能将注意力集中在它的解决方案上，后面你会看到，Haskell 结合事务内存将会给这个问题带来新的解决方案。不过此前还是让我们先来简单回顾一下传统的方案。

24.1.1 加锁的银行账户

目前，用于协调并发程序的主导技术仍是锁和条件变量。在一门面向对象的语言中，每个对象上都带有一个隐式的锁，而加锁则由 synchronized 方法来完成，但原理与经典的加锁解锁是一样的。在这样一门语言中，我们可以将银行账户类定义成下面这样：

```
class Account {
  Int balance;
  synchronized void withdraw( Int n ) {
    balance = balance - n; }
  void deposit( Int n ) {
    withdraw( -n ); }
}
```

withdraw 方法必须是 synchronized 的，这样两个线程同时调用它的时候才不会把 balance 减错了。synchronized 关键字的效果就等同于先对当前账户对象加锁，然后运行 withdraw 方法，最后再对当前账户对象解锁。

有了这个类之后，我们再来编写 transfer 转账函数：

```
void transfer( Account from, Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount ); }
```

对于非并发程序来说以上代码完全没问题，然而在并发环境中就不一样了，另一个线程可能会看到转账操作的“中间状态”：即钱从一个账户内被取走了，而同时另一个账户却还没收到这笔钱。值得注意的是，虽然 withdraw 和 deposit 这两个函数都是 synchronized，但这种情况还是会出现。在 from 上调用 withdraw 会将 from 加锁，执行提款操作，然后对其解锁。类似的，在 to 上调用 deposit 会将 to 加锁，执行存款操作，然后对其解锁。然而，关键是在这两个调用之间有一个间隙，在这个状态下待转账的钱既不在 from 账户中也不在 to 账户中。

在一个金融程序中，这种情况可能是无法容忍的。那我们该如何解决这个问题呢？通常的方案可能是在外面再包一层显式的加锁解锁操作，如下：

```
void transfer( Account from, Account to, Int amount ) {
  from.lock(); to.lock();
    from.withdraw( amount );
    to.deposit( amount );
  from.unlock(); to.unlock() }
```

但这种做法有一个致命的缺陷：它可能会导致死锁。我们考虑这样一种情况：在一个线程将一笔钱从 A 账户转到 B 账户的同时，另一个线程也正在将一笔钱从 B 账户转到 A 账户（当然，发生这种事情的几率很小）。这时便可能会出现两个线程各锁一个账户并都在等着对方释放另一账户的情况。

问题找出来了（不过，并发环境下的问题可不总是像这么容易就能找出来的），标准的解决方案是规定一个全局统一的锁顺序，并按照递增顺序来进行加锁。采用这种做法的代码如下：

```
if from < to
  then { from.lock(); to.lock(); }
  else { to.lock(); from.lock(); }
```

这个方法是可行的，但前提是必须事先知道要对哪些锁进行加锁，而后面这个条件并不是总能满足的。例如，假设 from.withdraw 的实现当账户余额不足时就会从 from2 上提款。遇到这种情况除非等到我们从 from 中提了钱否则是无法知道是否该对 from2 加锁的，而另一方面，一旦已经从 from 中提了钱，再想按“正确”顺序加锁便不可能了。更何况 from2 这个账户可能根本就只应对 from 可见，而不应被 transfer 知道。而且退一步说，就算 transfer 知道 from2 的存在，现在需要加的锁也已经由两个变成了三个（事先还要记得将这些锁正确排序）。

还有更糟的，如果我们需要在某些情况下阻塞（block），情况就会更加复杂。例如，要求 transfer 在 from 账户内余额不足的时候阻塞。这类问题通常的解决办法是在一个条件变量上等待，并同时释放 from 的锁。但更进一步，如果要求当 from 和 from2 中的总余额不够的时候阻塞呢？

24.1.2 “生锈”的锁

简而言之，在如今的并发编程领域占主导地位的技术，锁和条件变量，从根本上是有缺陷的。以下便是基于锁的并发编程中的一些公认的困难（其中有些我们在前文的例子中已经看到过了）。

锁加少了

> 容易忘记加锁，从而导致两个线程同时修改同一个变量。

锁加多了

> 容易加锁加得过多了，结果（轻则）妨碍并发，（重则）导致死锁。

锁加错了

> 在基于锁的并发编程中，锁和锁所保护的数据之间的联系往往只存在于程序员的大脑里，而不是显式地表达在程序代码中。结果就是一不小心便会加错了锁。

加锁的顺序错了

> 在基于锁的并发编程中，我们必须小心翼翼地按照正确的顺序来加锁（解锁），以避免随时可能会出现的死锁；这种做法既累人又容易出错，而且，有时候极其困难。

错误恢复

> 错误恢复也是个很麻烦的问题，因为程序员必须确保任何错误都不能将系统置于一个不一致的、或锁的持有情况不确定的状态下。

忘记唤醒和错误的重试

> 容易忘记叫醒在条件变量上等待的线程；叫醒之后又容易忘记重设条件变量。

然而，基于锁的编程，其最根本的缺陷，还是在于锁和条件变量不支持模块化编程。这里“模块化编程”是指通过粘合多个小程序来构造大程序的过程。而基于锁的并发程序是做不到这一点的。还是拿前面的例子来说吧：虽然 withdraw 和 deposit 这两个方法都是并发正确的，但如果原封不动的话，你能直接用它们实现出一个 transfer 来吗？不能，除非让锁协议暴露出来。而且遇到选择和阻塞的话还会更头疼。例如，假设 withdraw 在账户余额不足的情况下会阻塞。你就会发现，除非暴露锁条件，否则你根本无法直接利用 withdraw 函数从“A 账户或 B 账户（取决于哪个账户有足够余额）”进行提款；而且就算知道了锁条件，事情仍还是麻烦。另一些文献中也对锁并发的这种困难作了论述。[4]

[4] Edward A. Lee, "The problem with threads,"IEEE Computer, Vol. 39, No. 5, pp. 33－42, May 2006; J. K. Ousterhout, "Why threads are a bad idea (for most purposes)," Invited Talk, USENIX Technical Conference, January 1996; Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, "Composable memory transactions," ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05), June 2005.

24.2 软件事务内存

软件事务内存（STM）是迎接并发挑战的一种很有前途的新技术，本节将会详细说明这一点。我选用 Haskell 来介绍 STM，Haskell 是我见过的最美丽的编程语言，而且 STM 能够特别优雅地融入到 Haskell 中。如果你还不了解 Haskell，别担心，边看边学。

24.2.1 Haskell 中的副作用（Side Effects）和输入/输出（I/O）

Haskell 中的 transfer 函数写出来就像这样：

```
transfer :: Account -> Account -> Int -> IO ( )
```

```
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount = ...
```

以上代码的第二行，即以“--”开头的那行，是一个注释。代码的第一行是对transfer的函数类型的声明（类型声明以“::”前导）[5]，“Account -> Account -> Int -> IO ()”读作“接受一个Account，加上另一个Account（两个Account，代表转账的源账户和目标账户），以及一个Int（转账的数额），返回一个IO()类型的值”。最后一个类型（即返回类型）“IO()”说的是“transfer函数返回的是一个动作（action），这个动作被执行的时候可能会产生副作用（side effects），并返回一个‘()’类型的值”。“()”类型读作“单元（unit）”，该类型只有一个可能的值，也写作“()”，有点类似于C里面的void。transfer将IO()作为返回类型说明了执行过程中的副作用是我们调用transfer的惟一原因。那么，在介绍下面的内容之前，我们就必须首先知道Haskell是怎么对待副作用的。

[5]你可能会觉得奇怪，为什么在这个类型签名里面有三个“->”（难道不应该是一个吗——位于参数类型与返回类型之间？）其实这是因为Haskell支持所谓的currying，后者在任何介绍Haskell的书（比如Haskell: The Craft of Functional Programming, by S.J. Thompson [Addison-Wesley]）中或wikipedia上都能见到它的踪影。但currying不是本章要讲的重点，你大可以忽略除了最后一个“->”之外的所有“->”，即除了最后一个类型之外，其他都是函数的参数类型。

那么副作用是什么呢？副作用就是我们读写可变（mutable）状态所造成的影响（effect）。输入/输出是副作用的绝佳范例。例如，下面是两个Haskell函数，它们都具有输入/输出副作用：

```
hPutStr  :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
```

任何类型形如“IO t”（其中t可以为()，也可以为其他类型，如String）的值都是一个动作(action)。也就是说，在上面的例子中，(hPutStr h “hello”)是一个动作[1]，执行这个动作的效果便是在句柄h上输出“hello”[2]。类似的，(hGetLine h)也是一个动作，当它被执行的时候就会从句柄h代表的输入设备中读入一行输入并将其作为一个String返回。此外，利用Haskell的do关键字，我们可以将几个小的带副作用的程序“粘合”成一个大的。例如，下面的hEchoLine函数读入一个串并将它打印出来：

[1]在Haskell里面调用一个函数很简单，只须将函数名和它的各个参数并排写在一块儿就行了。在大多数语言中你都需要写成hPutStr(h, “hello”)，但Haskell里面只要写成(hPutStr h “hello”)就行了。

[2]Haskell中的句柄相当于C里面的文件描述（file descriptor）：指明对哪个文件或管道进行读写。跟Unix里面一样，Haskell里面也预定义了三个句柄：stdin、stdout和stderr。

```
hEchoLine :: Handle -> IO String
hEchoLine h = do { s <- hGetLine h
```

```
                    ; hPutStr h ("I just read: " ++ s ++ "\n")
                    ; return s }
```

do {a_1; …; a_n}结构将数个较小的动作（a_1…a_n）粘合成一个较大的动作。因此在上面的代码中，hEchoLine h 这个动作被执行的时候便会首先调用 hGetLine h 来读取一行输入，并将这行输入命名为 s。接着调用 hPutStr，将 s 加上前导的“I just read: ”[3]一并打印出来。最后串 s 被返回。最后一行 return s 比较有趣，因为 return 并非像在其他命令式语言中那样是语言内建的操作，而只是一个普普通通的函数，其函数类型如下：

[3]++操作符的作用是将两个串拼接起来。

```
return :: a -> IO a
```

也就是说，像 return v 这样一个操作被执行的时候，将会返回 v，同时并不会导致任何副作用[9]。return 函数可以作用于任何类型的值，这一点体现在其函数类型中：a 是一个类型变量，代表任何类型。

[9]“IO”的意思是一个函数可能有副作用，但并不代表它就一定会带来副作用。

输入/输出是一类重要的副作用。还有一类重要的副作用便是对可变（mutable）变量的读写。例如，下面这个函数将一个可变变量的值增 1：

```
incRef :: IORef Int -> IO ( )
incRef var = do { val <- readIORef var
                ; writeIORef var (val+1) }
```

incRef var 是一个动作。它首先执行 readIORef var 来获得变量 var 的值，并将该值绑定到 val;接着它调用 writeIORef 将 val+1 写回到 var 里面。readIORef 和 writeIORef 的类型如下：

```
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ( )
```

类型形如 IORef t 的值相当于一个指针或引用，指向或引用一个 t 类型的可变值（类似于 C 里面的 t*）。具体到 incRef，其参数类型为 IORef Int，因为 incRef 只操作 Int 型变量。

现在，我们已经知道了如何将数个较小的动作组合成一个较大的——但一个动作到底如何才算被真正调用呢？在 Haskell 里，整个程序其实就是一个 IO 动作，名叫 main。要运行这个程序，我们只需执行 main。例如下面就是一个完整的程序：

```
main :: IO ( )
```

```
main = do { hPutStr stdout "Hello"
          ; hPutStr stdout " world\n" }
```

该程序是一个顺序式（sequential）程序，因为 do 块将两个 IO 动作按顺序连接了起来。另一方面，要构造并发程序的话，我们便需要另一个原语（primitive）：forkIO：

```
forkIO :: IO a -> IO ThreadId
```

forkIO 是 Haskell 内建的函数，它的参数是一个 IO 动作，forkIO 所做的事情就是创建一个并发的 Haskell 线程来执行这个 IO 动作。一旦这个新线程建立，Haskell 的运行时系统便会将它与其他 Haskell 线程并行执行。例如假设我们将前面的 main 函数修改为[1]：

[10]其实，main 的第一行我们本可以写成“tid <- forkIO(hPutStr …)”的，这行语句会把 forkIO 的返回值（一个 ThreadId）绑定到 tid。然而由于本例中我们并不使用返回的 ThreadId，所以就把“tid <-”省略了。

```
main :: IO ( )
main = do { forkIO (hPutStr stdout "Hello")
          ; hPutStr stdout " world\n" }
```

现在，这两个 hPutStr 操作便能够并发执行了。至于哪个先执行（从而先打印出它的字符串）则是不一定的。Haskell 里面由 forkIO 产生出来的线程是非常轻量级的：只占用几百个字节的内存，所以一个程序里面就算产生上千个线程也是完全正常的。

读到这里，你可能会觉得 Haskell 实在是门又笨拙又麻烦的语言，incRef 的三行代码说穿了就做了 C 里面的一个 x++而已！没错，在 Haskell 里面，实施副作用的方式是非常显式且冗长的。然而别忘了，首先 Haskell 主要是一门函数式编程语言。大多数代码都是在 Haskell 的函数式内核里写的，后者的特点是丰富、高表达力、简洁。因而 Haskell 编程的精神就是“有节制地使用副作用”。

其次我们注意到，在代码中显示声明副作用的好处是代码能够携带许多有用的信息。考虑下面两个函数：

```
f :: Int -> Int
g :: Int -> IO Int
```

通过它们的类型我们便可以一眼看出，f 是一个纯函数，无副作用。给它一个特定的数（比如 42），那么每次对它的调用（f 42）都会返回同样的结果。相比较之下，g 就具有副作用了——这一点明明白白地显示在它的类型中。对 g 的不同的调用，就算参数相同，也可能会得到不同的值，因为，比如说，它可以通过 stdin 读取数据，或对一个可变变量进行修改。后面你会发现，这种显式声明副作用的做法其实非常有用。

最后，前面提到的所谓动作（action，比如 I/O 动作）其实本身也是值（Haskell 中的动作也是一等公民）：它们也可以被作为参数传递或作为返回值返回。比如下面就是一个纯粹用 Haskell 写的（而非内建的）模拟（简化的）for 循环的函数：

```
nTimes :: Int -> IO ( ) -> IO ( )
nTimes 0 do_this = return ( )
nTimes n do_this = do { do_this; nTimes (n-1) do_this }
```

这是一个递归函数，其第一个参数是一个 Int，表示要循环多少次，第二个参数则是一个动作（action）：do_this；该函数返回一个动作，后者被执行的时候会把 do_this 重复做 n 遍。比如下面这段代码利用 nTimes 来重复输出 10 个“Hello”：

```
main = nTimes 10 (hPutStr stdout "Hello\n")
```

这其实从效果上就等同于允许用户自定义程序的控制结构了。

话说回来，本章的目的并不是要对 Haskell 作一个全面的介绍，而且即便是对于 Haskell 里面的副作用我们也只是稍加阐述。如果你想进一步了解的话，可以参考我写的一篇指南“Tackling the awkward squad”[11]。

[11] Simon Peyton Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell," C. A. R. Hoare, M. Broy, and R. Steinbrueggen, editors, Engineering theories of software construction, Marktoberdorf Summer School 2000, NATO ASI Series, pp. 47－96, IOS Press, 2001.

24.2.2 Haskell 中的事务

OK，终于可以回到我们的 transfer 函数了。其代码如下：

```
transfer :: Account -> Account -> Int -> IO ( )
-- Transfer 'amount' from account 'from' to account 'to'
transfer from to amount
 = atomically (do { deposit to amount
                  ; withdraw from amount })
```

里面的那个 do 块你应该不觉得陌生了吧：它先是调用 deposit 将 amount 数目的钱存入 to 账户，然后再从 from 账户中提取 amount 数目的钱。至于 deposit 和 withdraw 这两个辅助函数我们待会再来写，现在我们先来看看对 atomically 的调用。atomically 的参数是一个动作，它会将该动作作为一个原子来执行。更精确地说，atomically 有如下两个特性作保证：

原子性

> atomically act 调用所产生的副作用对于其他线程来说是“原子的”。这就保证了另一个线程不可能观察到钱被从一个账户中取出而同时又没来得及存入另一个账户中去的中间状态。

隔离性

> 在 atomically act 执行的过程中，act 这个动作与其他线程完全隔绝，不受影响。这就好像在 act 开始执行的时候世界停顿了，直到 act 执行完毕之后世界才又开始恢复运行。

至于 atomically 函数的执行模型，简单的做法是：存在一个惟一的全局锁；atomically act 首先获取该锁，然后执行动作 act，最后释放该锁。这个实现虽然保证了原子性，但粗暴地禁止了任意两个原子块在同一时间执行。

上面说的这个模型有两个问题。第一，它并没有保证隔离性：比如一个线程在执行一个原子块的过程中访问了一个 IORef（此时该线程持有全局锁），另一个线程此时照样可以直接对同一个 IORef 进行写操作（只要这个写操作不在原子块内）。这就破坏了隔离性保证。第二，它极大的损害了执行性能，因为即便各个原子块之间互不相干，也必须被串行化执行。

第二个问题待会在“事务内存实现”一节会详细讨论。目前先来看第一个问题。第一个问题可以通过类型系统轻易解决。我们将 atomically 函数赋予如下类型：

```
atomically :: STM a -> IO a
```

atomically 的参数是一个类型为 STM a 的动作。STM 动作类似于 IO 动作，它们都可能具有副作用，但 STM 动作的副作用的容许范围要小得多。STM 中你可以做的事情主要就是对事务变量（类型为 TVar a）进行读写，就像我们在 IO 动作里面主要对 IORef 进行读写一样[12]。

[12]这儿其实有一个命名上的不一致：STM 变量被命名为 TVar，然而普通变量却被命名为 IORef——其实要么应该是 TVar/IOVar，要么应该是 TRef/IORef 才对。但事到如今已经没法再改了。

```
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ( )
```

跟 IO 动作一样，STM 动作也可以由 do 块组合起来，实际上，do 块针对 STM 动作进行了重载，return 也是；这样它们便可以运用于 STM 和 IO 两种动作了[13]。例如，下面是 withdraw 的代码：

[13]其实 Haskell 并没有特别针对 IO 和 STM 动作来重载 do 和 return，IO 和 STM 其实只是一个更一般的模式的特例，这个更一般的模式便是所谓的 monad（P. L. Wadler 在“The essence of functional programming” 20th ACM Symposium on Principles of Programming Languages [POPL '92], Albuquerque, pp. 1－14, ACM, January 1992 中有描述），do 和 return 的重载便是通过用 Haskell 的非常泛化的“类型的类型”（type-class）系统来表达 monad 而得以实现的（described in P. L. Wadler and S. Blott, "How to make

ad-hoc polymorphism less ad hoc," Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, ACM, January 1989; and Simon Peyton Jones, Mark Jones, and Erik Meijer, "Type classes: an exploration of the design space," J. Launch-bury, editor, Haskell workshop, Amsterdam, 1997）。

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ( )
withdraw acc amount
  = do { bal <- readTVar acc
       ; writeTVar acc (bal - amount) }
```

我们用一个包含一个 Int（账户余额）的事务变量来表示一个账户。withdraw 是一个 STM 动作，将账户中的余额提走 amount。

为了完成 transfer 的定义，我们可以通过 withdraw 来定义 deposit：

```
deposit :: Account -> Int -> STM ( )
deposit acc amount = withdraw acc (- amount)
```

注意，transfer 从根本上执行了四个基本的读写操作：对 to 账户的一次读和一次写；以及对 from 账户的一次读和一次写。这四个操作是被当成一个原子来执行的，其执行满足本节（“一个简单的例子：银行账户”）开头的要求。

Haskell 的类型系统优雅地阻止了我们在事务之外读写 TVar。例如假设我们这样写：

```
bad :: Account -> IO ( )
bad acc = do { hPutStr stdout "Withdrawing..."
             ; withdraw acc 10 }
```

以上代码不能通过编译，因为 hPutStr 是一个 IO 动作，而 withdraw 则是一个 STM 动作，这两者不能放在同一个 do 块中。但如果我们把 withdraw 再放在一个 atomically 调用当中就可以了，如下：

```
good :: Account -> IO ( )
good acc = do { hPutStr stdout "Withdrawing..."
              ; atomically (withdraw acc 10) }
```

24.2.3 事务内存实现

有了前面提到过的原子性和隔离性保证，我们其实便可以放心使用 STM 了。不过我常常还是觉得一个合理的实现模型会给直觉理解带来很大的帮助，本节就来介绍这么一个实现模型。但要注意的是，这只是所有可能实现中的一种。STM 抽象

的一个漂亮之处就在于它提供了一个小巧干净的接口，而实现这个接口可以有多种方式，可简单可复杂。

在可行的实现方案中，有一个方案特别吸引人，那就是在数据库实现里被采用的所谓的“乐观执行（optimistic execution）”。当 atomically act 被执行的时候，Haskell 运行时系统会为它分配一个线程本地的事务日志，该日志最初的时候是空的，随着 act 动作被一步步执行（其间并不加任何形式的锁），每次对 writeTVar 的调用都会将目标 TVar 变量的地址和新值写入日志；而并不是直接写入到那个 TVar 变量本身。每次对 readTVar 的调用都会首先寻找日志里面有没有早先被写入的新值，没有的话才会从目标 TVar 本身读取，并且，在读取的时候，一份拷贝会被顺便存入到日志中。同一时间，另一个线程可能也在运行着它自己的原子块，疯狂地读写同样一组 TVar 变量。

在 act 这个动作执行完毕之后，运行时系统首先会对日志进行验证，如果验证成功，就会提交（commit）日志。那么验证是怎么进行的呢？运行时系统会检查日志中缓存的每个 readTVar 的值是否与它们对应的真正的 TVar 相匹配。是的话便验证成功，并将日志中缓存的写操作结果全都提交到相应的 TVar 变量上。

必须注意的是，以上验证-提交的整个过程是完全不可分割的：底层实现会将中断禁止掉，或使用锁或 CAS（compare-and-swap）指令等任何可行的方法来确保这个过程对于其他线程来说就像“一瞬间”的事情一样。但由于所有这些底层工作都由实现来完成，所以程序员不用担心也不用考虑它是怎么完成的。

一个自自而然的问题是：如果验证失败呢？如果验证失败，就代表该事务看到的是不一致的内存视图。于是事务被中止（abort），日志被重新初始化，然后整个事务从头再来过。这个过程就叫做重新执行（re-execution）。由于此时 act 动作的所有副作用都还没有真正提交到内存中，因此重新执行它是完全没问题的。然而有一点必须注意：act 不能包含任何除了对 TVar 变量读写之外的副作用，比如下面这种情况：

```
atomically (do { x <- readTVar xv
               ; y <- readTVar yv
               ; if x>y then launchMissiles
                        else return () })
```

launchMissiles::IO ()这个函数的副作用是“头晕、恶心、呕吐”。由于这个原子块执行的时候并没有加锁，所以如果同时有其他线程也在修改变量 xv 和 yv 的话，该线程就可能观察到不一致的内存视图。而一旦这种情况发生，发射导弹（launchMissiles）可就闯了大祸了，因为等到导弹发射完了才发现验证失败就已经来不及了。不过幸运的是，Haskell 的类型系统会阻止冒失的程序员把 IO 动作（比如这个 launchMissiles）放在 STM 动作中执行，所以，以上代码会被类型系统阻止。这从另一个方面显示了将 IO 动作跟 STM 动作区分开来的好处。

24.2.4 阻塞和选择

到目前为止我们介绍的原子块从根本上还缺乏一种能力：无法用来协调多个并发线程。这是因为还有两个关键的特性不具备：阻塞和选择。本节就来介绍如何扩充基本的 STM 接口从而使之包含以上两个特性（当然，在完全不破坏模块性的前提下）。

假设当一个线程试图从一个账户中提取超过账户余额的钱时这个线程便会阻塞。并发编程中这类情况很常见：例如一个线程在读取到一个空的缓冲区时阻塞；或在等待一个事件的时候阻塞，等等。为了支持这种场景，我们往 STM 中加入 retry 功能，retry 的类型为：

```
retry :: STM a
```

以下是 withdraw 的一个修改过的版本，该版本当余额不足的时候便会转入到阻塞状态：

```
limitedWithdraw :: Account -> Int -> STM ( )
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; if amount > 0 && amount > bal
         then retry
         else writeTVar acc (bal - amount) }
```

retry 的语意很简单：当一个 retry 语句被执行的时候，当前事务便被“丢弃”并等待某个时候再重新执行。当然，这里的“某个时候”可以是立即，但这样做不够高效：如果重新执行的时候账户余额根本没有改变的话还是白搭，结果又是 retry。一个高效的实现不会这么做，而是会一直阻塞，直到有其他线程对 acc 进行了写操作（译注：即改变了账户余额）。那么，问题是实现怎么知道要在 acc 变量上等待呢？很简单，因为该事务在到达 retry 这一点之前的执行路径上读取的就是 acc，而这个读取动作会被事务日志完完整整地记录下来（译注：所以只要往日志里“瞄一眼”就知道了）。

limitedWithdraw 中的条件有一个非常普遍的模式：检查一个布尔条件是否满足，如果不满足则 retry。这个模式很容易抽象出来做成一个函数：

```
check :: Bool -> STM ( )
check True = return ( )
check False = retry
```

利用这个 check 函数来重新表达 limitedWithdraw 如下(是不是简洁了一些？)：

```
limitedWithdraw :: Account -> Int -> STM ( )
limitedWithdraw acc amount
  = do { bal <- readTVar acc
       ; check (amount <= 0 || amount <= bal)
```

```
        ; writeTVar acc (bal - amount) }
```

接下来我们考虑选择（choice）。假设你想要从 A 账户上取钱，但前提是 A 上必须要有足够的钱，如果没有的话，你便改从 B 上取。为此我们必须实现以下能力：如果前一条路径需要 retry，则选择另一条路径。于是 STM Haskell 加入了另一个原语：orElse，来支持选择。orElse 的类型为：

```
orElse :: STM a -> STM a -> STM a
```

跟 atomically 函数一样，orElse 的参数也是动作，orElse 将小的动作粘合成较大的动作。其语意如下：(orElse a1 a2)首先会执行动作 a1，如果 a1 发生了 retry，那么它便会转而执行 a2。如果 a2 也 retry 了，那么整个动作（a1 和 a2）便被 retry。orElse 的用法很简单：

```
limitedWithdraw2 :: Account -> Account -> Int -> STM ( )
-- (limitedWithdraw2 acc1 acc2 amt) withdraws amt from acc1,
-- if acc1 has enough money, otherwise from acc2.
-- If neither has enough, it retries.
limitedWithdraw2 acc1 acc2 amt
  = orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

由于 orElse 返回的也是一个 STM 动作，因此我们便可以将 orElse 调用的结果给另一个 orElse，如此嵌套，便可以实现任意数目的备选路径。

24.2.5 基本 STM 操作小结

本节我们介绍了 STM Haskell 支持的所有关键的事务内存操作。表 24-1 是一个小结。注意，里面有一个操作：newTVar 到目前为止我们还没有提到。newTVar 是用来创建新的 TVar 变量的，下一节我们会用到它。

表 24-1 STM Haskell 支持的关键操作

操作	类型签名
atomically	STM a -> IO a
retry	STM a
orElse	STM a -> STM a -> STM a
newTVar	a -> STM (TVar a)
readTVar	TVar a -> STM a
writeTVar	TVar a -> a -> STM ( )

24.3 圣诞老人问题

本节将为你展示一个完整的、可运行的 STM 程序。一个众所周知的例子便是所谓的“圣诞老人问题”[1]，这个问题最初由 Trono 提出[2]：

[1] My choice was influenced by the fact that I am writing these words on December 22.

[2] J. A. Trono, "A new exercise in concurrency," SIGCSE Bulletin, Vol. 26, pp. 8－10, 1994.

问题描述是这样的：圣诞老人总是在睡觉，直到被他的（放假归来的）九头驯鹿中的任意一头叫醒，或被他的十个小矮人中的任意三个（一组）叫醒。如果是被驯鹿叫醒的，他就把驯鹿们套上雪橇出门去给小朋友们送礼物，回来之后再解开驯鹿（放假）。如果是被一组（三个）小矮人叫醒的，他就将这三个小矮人一个个带进书房，跟他们交流玩具的制造，然后再将他们一个个领出去（好让他们回去继续工作）。如果发现既有一组小矮人又有一群驯鹿在等他的话，圣诞老人便优先选择驯鹿。

使用一个众所周知的例子的好处便是在一些其他语言中已经有了描述得很好的解决方案了，这样你就能够将我们马上要介绍的方案跟其他语言中既有的方案进行一目了然的比较。值得注意的是，Trono 的论文中给出了一个基于信号量的方案，那个方案只是部分正确的；Ben-Ari 用 Ada95 和 Ada 给出了解决方案[1]；Benton 也用 Polyphonic C#（译注：C#的一个扩展，主要加入基于 Join Calculus 的并发编程模型）写了一个解决方案[2]。

[15] Nick Benton, "Jingle bells: Solving the Santa Claus problem in Polyphonic C#," Technical report, Microsoft Research, 2003.

[16] Mordechai Ben-Ari, "How to solve the Santa Claus problem," Concurrency: Practice and Experience, Vol. 10, No. 6, pp. 485－496, 1998.

24.3.1 驯鹿和小矮人

用 STM Haskell 来解决这个问题，基本理念是这样的：圣诞老人分别给小矮人和驯鹿各创建一个“群”。每个小矮人（或驯鹿）都试图去加入相应的群。如果成功的话，便得到两扇“门”。第一扇门允许圣诞老人控制什么时候让小矮人们进入书房，并得知什么时候他们全都进去了。类似的，第二扇门则控制小矮人们离开书房。圣诞老人等待他创建的任何一个群准备好，并用那个准备好的群的两扇门来控制他们的小帮手们（小矮人或驯鹿）完成工作。也就是说，不管是驯鹿还是小矮人，他们一直都在做一个无限循环：试图加入群——在圣诞老人的看管下穿过“门”——等待一段时间（比如驯鹿便会去休假）——再次试图加入群。

用 Haskell 来描述上述逻辑，我们便得到如下的代码（这是针对小矮人的）[1]：

[17]为什么是 elf1 而不是 elf，是因为这个函数只做一重循环，而实际上小矮人会不断重复加入群。后面我们会基于 elf1 来定义 elf。

```
elf1 :: Group -> Int -> IO ( )
```

```
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group
                       ; passGate in_gate
                       ; meetInStudy elf_id
                       ; passGate out_gate }
```

elf1 的参数是一个“群”（Group）以及一个 Int，后者惟一指代该小矮人的身份。这个 Int 只在调用 meetInStudy 的时候用到，meetInStudy 简单地打印出一行消息表明正在发生的事情[18]：

[18]putStr 是一个库函数，它会调用 hPutStr stdout。

```
meetInStudy :: Int -> IO ( )
meetInStudy id = putStr ("Elf " ++ show id ++ " meeting in the study\n")
```

小矮人调用 joinGroup 加入群，然后调用 passGate 来穿过门，这两个函数如下：

```
joinGroup :: Group -> IO (Gate, Gate)
passGate  :: Gate -> IO ( )
```

驯鹿们的代码几乎完全一样，惟一的区别就是驯鹿的任务是送礼物而不是进书房讨论：

```
deliverToys :: Int -> IO ( )
deliverToys id = putStr ("Reindeer " ++ show id ++ " delivering toys\n")
```

由于 IO 动作也是值，所以我们便可以将驯鹿和小矮人们的逻辑抽象出一个公共模式来，如下：

```
helper1 :: Group -> IO () -> IO ( )
helper1 group do_task = do { (in_gate, out_gate) <- joinGroup group
                           ; passGate in_gate
                           ; do_task
                           ; passGate out_gate }
```

helper1 的第二个参数是一个 IO 动作，代表待执行的任务。helper1 负责将这个任务放在两个 passGate 调用之间执行。有了这个辅助函数，我们便可以基于它来定义小矮人和驯鹿函数了：

```
elf1, reindeer1 :: Group -> Int -> IO ( )
elf1      gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

24.3.2 门和群

先来看“门”这个抽象概念。“门”支持如下接口：

```
newGate     :: Int -> STM Gate
passGate    :: Gate -> IO ( )
operateGate :: Gate -> IO ( )
```

每个门都有一个固定的容限 n。这个 n 是在我们新建门的时候指定的。此外门还有一个剩余容许量 m，这是一个可变的（mutable）变量。passGate 被调用的时候，m 就会减一。如果剩余容许量减至零，那么对 passGate 的调用就会阻塞。一个门最初被创建起来的时候剩余容许量为 0，因此任何人都不能进入。而圣诞老人则通过调用 operateGate 来打开这扇门，operateGate 会将门的剩余容许量置为 n。

下面便是门（Gate）的一个可能实现：

```
data Gate = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0; return (MkGate n tv) }

passGate :: Gate -> IO ( )
passGate (MkGate n tv)
  = atomically (do { n_left <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ( )
operateGate (MkGate n tv)
  = do { atomically (writeTVar tv n)
       ; atomically (do { n_left <- readTVar tv
                        ; check (n_left == 0) }) }
```

第一行是类型声明，声明 Gate 为一个新的数据类型，并具有一个数据构造子叫 MkGate[19]，该构造子有两个成员：一个 Int 型数据，表示该门的容限。另一个则是一个 TVar，表示在门关闭之前还有多少人可以穿过它。如果该 TVar 为 0，就表明门是关闭的。

[19]Haskell 中的类型声明不像 C 里面的结构声明，MkGate 只是一个结构 tag。

函数 newGate 负责创建一个新的门，它先是分配一个 TVar，然后通过调用 MkGate 这个构造子来创建一个 Gate 对象。无独有偶，passGate 利用模式匹配来将 MkGate 构造子拆分开来（译注：即(MkGate n tv)），然后 passGate 将 Tvar 的值减一，

并利用 check 来确保 tv> 0。前面“阻塞和选择”一节我们实现 withdraw 的时候也用到过这个 check。最后是 operateGate 函数，operateGate 将门的最大容限值写入 Tvar，并等待 Tvar 被减至 0。

群（Group）的接口如下：

```
newGroup   :: Int -> IO Group
joinGroup  :: Group -> IO (Gate,Gate)
awaitGroup :: Group -> STM (Gate,Gate)
```

跟门类似，群刚创建起来的时候也是空的，并带有一个指定的容限。小矮人们可以通过调用 joinGroup 来加入群，如果群已满那么 joinGroup 就会阻塞。而圣诞老人则会调用 awaitGroup 来等待群被加满；群满了之后，圣诞老人就会通过 awaitGroup 的返回值得到该群对应的两扇门，然后群立即被用两扇新门重新初始化，好让另一组焦急的小矮人们开始集合。

下面是 newGroup 的一个可能的实现：

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))

newGroup n = atomically (do { g1 <- newGate n; g2 <- newGate n
                            ; tv <- newTVar (n, g1, g2)
                            ; return (MkGroup n tv) })
```

同样，Group 是一个新声明的数据类型，其构造子 MkGroup 具有两个成员：该 Group 的容限以及一个 TVar，后者包含该 Group 剩余的名额以及两个 Gate 对象。要创建一个新的 Group 对象（newGroup），先要创建两个 Gate 对象（g1，g2），并初始化一个新的 TVar（tv），然后调用 Group 的构造子 MkGroup。

而 joinGroup 和 awaitGroup 基本就是基于上面的这些数据结构来实现的：

```
joinGroup (MkGroup n tv)
  = atomically (do { (n_left, g1, g2) <- readTVar tv
                   ; check (n_left > 0)
                   ; writeTVar tv (n_left-1, g1, g2)
                   ; return (g1,g2) })

awaitGroup (MkGroup n tv)
  = do { (n_left, g1, g2) <- readTVar tv
       ; check (n_left == 0)
       ; new_g1 <- newGate n; new_g2 <- newGate n
       ; writeTVar tv (n,new_g1,new_g2)
       ; return (g1,g2) }
```

注意，awaitGroup 在重新初始化 Group 对象的时候会新建两个 Gate 对象。这确保了当圣诞老人在书房里讨论时，一个新群可以同时处于集结之中；如果不新建 Gate 对象的话，新群中的小矮人便可能会将旧群中打瞌睡的那些家伙给挤出去[1]。

[1] 假设旧群满了，此时 joinGroup 已返回给该群中的每个小矮人两扇门，而圣诞老人调用的 awaitGroup 随后也观察到群已满，于是打开门，同时群被重新开放，但注意，群上附着的两扇门还是原来的两扇，这就表示，当另一组小矮人调用 joinGroup 的时候得到的返回值仍然还是那两扇门，于是新群中的小矮人便和旧群中的小矮人们同挤一扇门，如果这时（在进门时）旧群中的某个小矮人不幸睡着了（比如线程时间片用完了），便会被新群中的小矮人捷足先登了——译者注。

回顾一下这节，你可能会注意到，有些对群和门的操作是 IO 类型的（比如 newGroup 和 joinGroup），而有些则是 STM 类型的（比如 newGate 和 awaitGroup）。那么为什么这么设置呢？就拿 newGroup 来说吧，newGroup 有一个 IO 型的操作，也就是说我们无法在 STM 动作中调用它。但实际上我将 newGroup 做成 IO 型只是为了方便起见：我本可以把 newGroup 定义中的 atomically 调用去掉的，这样 newGroup 便成了 STM 类型的了，但这么一来每次调用 newGroup 的时候我们便都需要手动将其包在 atomically 之中了（atomically(newGroup n)）。而另一方面，将 newGate 做成 STM 操作的好处是能让它的可组合性（composability）更好，只不过就本应用程序来说 newGroup 并不需要这个可组合性，所以我才将它做成 IO 的，然而。由于我想在 newGroup 中调用 newGate，因此 newGate 的可组合性便有意义了，这便是我将 newGate 设为 STM 操作的原因。

一般来说，在设计一个库的时候，你应当尽可能地把函数的类型设为 STM。你可以把 STM 动作看作组合积木，小的 STM 动作可以（通过 do {…}，retry，orElse）组合起来成为更大的 STM 动作。然而，一旦你将一个块用 atomically 包裹起来之后，它就变成了一个 IO 动作，就再也不能利用 atomically 跟其他动作组合起来了。但 IO 动作也有它自己的优点：一个 IO 动作可以执行任意的、不可撤销的输入/输出（比如 launchMissiles）。

[1]（参见 newGroup 的实现——译者注）

因此，好的库设计应当尽可能的暴露 STM 动作（而不是 IO 动作），因为 STM 动作是可组合的；它们的类型表明了它们不会执行不可撤销的操作。而另一方面，库的用户总是可以很容易地将 STM 动作包装成 IO 动作（外面加一层 atomically 调用即可），但反过来就不行了。

然而，有时候还是必须使用 IO 动作的。比如 operateGate。operateGate 中的两个对 atomically 的调用无法并成一个，因为第一个 atomically 调用具有一个外部可见的副作用（开门），而第二个 atomically 调用则需要等到所有的小矮人们都醒过来并穿过了这扇门之后才能执行结束，否则便会一直阻塞。[1]因此 operateGate 必须是 IO 类型的。

[1]所以，前一个 atomically 操作不生效，后一个 atomically 操作是不可能完成的——译者注。

24.3.3 主程序

我们首先把程序的骨架实现出来，注意，代表圣诞老人的函数（santa）还没有实现，但先不管它：

```
main = do { elf_group <- newGroup 3
          ; sequence_ [ elf elf_group n | n <- [1..10] ]

          ; rein_group <- newGroup 9
          ; sequence_ [ reindeer rein_group n | n <- [1..9] ]

          ; forever (santa elf_group rein_group) }
```

第一行创建了一个大小为 3 的群。第二行则稍微需要解释一下：它利用了所谓的列表内涵式（list comprehension）来创建一组 IO 动作，然后调用 sequence_来顺序执行它们。列表内涵式“[e|*x*<-*xs*]”读作“由一切 e 所构成的列表，其中 x 来自列表 xs”。因此本例中 sequence_的参数为：

```
[elf elf_group 1, elf elf_group 2, ..., elf elf_group 10]
```

这些调用每个都会返回一个 IO 动作，后者在被执行的时候会新建一个小矮人线程。而 sequence_函数则接受一组 IO 动作作为参数，返回的也是一个 IO 动作，后者被执行的时候会按列表中的顺序执行那组作为参数的 IO 动作[20]。

[20]类型“[IO a]”读作“一个由类型为 IO a 的值构成的列表”。另外你可能会奇怪 sequence_后面为什么要加上一个下划线，其实这是因为另外还有一个与它相关的函数也叫 sequence，这个 sequence 的类型则是[IO a] -> IO [a]，其功能是将一组动作执行后的结果收集到一个列表中。sequence 和 sequence_都定义在 Prelude 库中（Prelude 库是缺省导入的）。

```
sequence_ :: [IO a] -> IO ( )
```

elf 函数是基于 elf1 写的，但有两点区别。首先是我们想要 elf 重复执行，每重循环延迟一个不确定的时间间隔；其次，我们想要它在单独的线程中运行：

```
elf :: Group -> Int -> IO ThreadId
elf gp id = forkIO (forever (do { elf1 gp id; randomDelay }))
```

forkIO 将它的参数（一个动作）放在一个单独的 Haskell 线程中执行（见前面的小节“Haskell 中的副作用和输入/输出”）。forkIO 的实参是一个对 forever 的调用；forever，顾名思义，会将一个动作重复执行（一个与它类似但有微妙差别的函数是 nTimes，见“Haskell 中的副作用和输入/输出”）：

```
forever :: IO () -> IO ( )
-- Repeatedly perform the action
forever act = do { act; forever act }
```

最后，表达式(elf1 gp id)是一个 IO 动作，我们想要将它不确定地重复执行，每次执行随机延迟一段时间：

```
randomDelay :: IO ( )
-- Delay for a random time between 1 and 1,000,000 microseconds
randomDelay = do { waitTime <- getStdRandom (randomR (1, 1000000))
                 ; threadDelay waitTime }
```

主程序中剩下的部分含义就很明显了。创建 9 头驯鹿和创建 10 个小矮人的方式是一样的：

```
reindeer :: Group -> Int -> IO ThreadId
reindeer gp id = forkIO (forever (do { reindeer1 gp id;
randomDelay }))
```

主程序的最后一行代码利用 forever 来运行 santa。下面我们就来说说最后一个问题——圣诞老人（Santa）的实现。

24.3.4 圣诞老人的实现

圣诞老人是这个问题里面最有趣的，因为他会进行选择。他必须等到一组驯鹿或一组小矮人在那儿等他的时候才会继续行动。一旦他选择了是带领驯鹿还是小矮人之后，他便将他们带去做该做的事。圣诞老人的代码如下：

```
santa :: Group -> Group -> IO ( )
santa elf_gp rein_gp
  = do { putStr "----------\n"
     ; (task, (in_gate, out_gate))
            <- atomically (orElse
                   (chooseGroup rein_gp "deliver toys")
                   (chooseGroup elf_gp "meet in my study"))

     ; putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
     ; operateGate in_gate
      -- Now the helpers do their task
     ; operateGate out_gate }

where
  chooseGroup :: Group -> String -> STM (String, (Gate,Gate))
  chooseGroup gp task = do { gates <- awaitGroup gp
                           ; return (task, gates) }
```

圣诞老人进行选择的关键就在那个 orElse 上。orElse 首先会试图选择驯鹿（驯鹿优先），如果驯鹿没有准备好就选择小矮人。chooseGroup 会对相应的群调用 awaitGroup，并返回一个对偶（pair）“(task, gates)”，其中 task 是一个字符串，代表待执行的任务（“deliver toys”或“meet in my study”），gates 则本身又是一个对偶，它包含的是两扇门，圣诞老人通过操作这两扇门来带领一群小矮人或驯鹿完成任务。一旦在驯鹿和小矮人之间的选择完成了，圣诞老人便打印出一则消息表示待执行的任务，并依次操纵（operateGate）两扇门。

该实现工作起来自然是没问题的，但不妨让我们来看看另一个实现，这个实现更具一般性，因为圣诞老人的程序显示出了一个非常普遍的模式：一个线程（本例中是圣诞老人）在一个原子事务中作了一次选择，并根据选择的结果接着执行一个或多个事务。另一个典型的场景是：从多个消息队列中获取一则消息，并针对该消息做一些事情，然后重复这个过程。在圣诞老人问题里面，这里的后续操作对小矮人和对驯鹿基本是一样的——两种情况下圣诞老人都得打印一则消息并操纵两扇门。如果对小矮人的逻辑和对驯鹿的逻辑差别很大的话刚才上面那种做法就行不通了，一个补救的办法是使用一个布尔变量来表示到底选择了小矮人还是驯鹿，并根据具体选择了哪一方来决定做什么事情；但一旦选择的可能性多了，这种做法同样还是不够方便。下面是一个更好的解决方案：

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
  = do { putStr "----------\n"
       ; choose [(awaitGroup rein_gp, run "deliver toys"),
                 (awaitGroup elf_gp, run "meet in my study")] }
  where
    run :: String -> (Gate,Gate) -> IO ()
    run task (in_gate,out_gate)
      = do { putStr ("Ho! Ho! Ho! let's " ++ task ++ "\n")
           ; operateGate in_gate
           ; operateGate out_gate }
```

choose 函数就像一个“保险命令”一样：它接受一组对偶（pairs），等到某个对偶的第一个元素可以“开火”了，便执行其第二个元素。因此 choose 的类型如下[21]：

[21]在 Haskell 中，类型[*ty*]的意思是一个元素类型为 ty 的列表。本例中 choose 的参数为一个由对偶((*ty1*, *ty2*))构成的列表；其中对偶的第一个元素的类型为 STM a，第二个元素则是一个函数，类型为 a->IO()。

```
choose :: [(STM a, a -> IO ())] -> IO ( )
```

刚才提到“保险命令”的比喻，这里的“保险”就是对偶的第一个元素，即一个 STM 动作，返回类型为 a；当这个 STM 动作“准备好了”（不发生 retry）之后，choose 便可以将其返回值传给对偶的第二个元素，当然，后者必须是一个函数，且接受一个 a 类型的参数。了解了以上这些之后再来阅读 santa 的代码就应该毫无困难了。santa 利用 awaitGroup 来等待一个群准备好；choose 拿到 awaitGroup

返回的两扇门之后便将它们传给 run 函数，后者依次操纵这两扇门——operatorGate 会一直阻塞，直到所有小矮人（或驯鹿）都穿过门之后才会返回。

choose 的代码虽然只有寥寥数行，但要真正弄明白它还是得费点脑筋的：

```
choose :: [(STM a, a -> IO ( ))] -> IO ( )
choose choices = do { act <- atomically (foldr1 orElse actions)
                    ; act }
  where
    actions :: [STM (IO ( ))]
    actions = [ do { val <- guard; return (rhs val) }
              | (guard, rhs) <- choices ]
```

首先来看 actions，actions 是一个列表，它的每个元素都是一个 STM 动作。choose 将 actions 和 orElse 用 foldr1 结合起来（foldr1 orElse [x1, … , xn]的结果是 x1 orElse x2 orElse x3 … orElse xn）。这些 STM 动作（即 actions 列表里面的元素）每个又都返回一个 IO 动作（所以才有 STM(IO( ))这个类型），后者也就是它一旦被选中之后要做的事情。choose 首先在各个动作之间作一次原子选择，取得返回出来的动作（act，类型为 IO( )），然后执行该动作。而列表 actions 又是如何构造出来的呢？答案是只需对 choices 列表里面的每个对偶(guard, rhs)，运行 guard（一个 STM 动作），再将 guard 的返回值作为参数交给 rhs，并返回后者执行的结果即可。

24.3.4 编译并运行程序

以上便是这个例子的全部代码。要运行它，只需在程序开头再添上几个 import 语句即可[1]：

[1]代码也可以在这里下载到：http://research.microsoft.com/~simonpj/papers/stm/Santa.hs.gz

```
module Main where
  import Control.Concurrent.STM
  import Control.Concurrent
  import System.Random
```

使用 GHC（Glasgow Haskell Compiler）编译代码[2]：

[23]GHC 是免费的，在这里下载：http://haskell.org/ghc。

```
$ ghc Santa.hs -package stm -o santa
```

最后运行：

```
$ ./santa
----------
```

```
Ho! Ho! Ho! let's deliver toys
Reindeer 8 delivering toys
Reindeer 7 delivering toys
Reindeer 6 delivering toys
Reindeer 5 delivering toys
Reindeer 4 delivering toys
Reindeer 3 delivering toys
Reindeer 2 delivering toys
Reindeer 1 delivering toys
Reindeer 9 delivering toys
----------
Ho! Ho! Ho! let's meet in my study
Elf 3 meeting in the study
Elf 2 meeting in the study
Elf 1 meeting in the study
...and so on...
```

24.4 对 Haskell 的一些思考

Haskell 首先是一门函数式编程语言，但我觉得它同样也是世界上最漂亮的命令式语言。如果把 Haskell 当成一门命令式语言来看待的话，我们会发现，它具有一些不寻常的特性：

- 动作（有副作用）和纯值（无副作用）被严格区分开来。
- 动作也是真正的值。它们可以被作为参数、作为返回值、放在列表内等等，所有这些操作都不会带来副作用[1]。

[1]（因为并不会导致动作被执行——译者注

利用动作作为第一类值，我们便可以借助于它来定义应用相关的控制结构，而不是受制于语言设计者定义的那一套。例如，nTimes 就是一个简单的 for 循环结构。而刚才提到的 choose 则实现了一种可以称之为“保险命令”的功能。动作也是值的这一性质还有许多其他应用。在 main 函数中，我们利用 Haskell 丰富的表达力（列表内涵式）生成了一列动作，然后再利用 sequence_来依次执行这些动作。同样，前面在定义 helper1 的时候，我们也是利用的这一性质：我们从一块代码中抽象出了一个动作，从而提升了代码的模块性。当然，圣诞老人的实现代码量本就不多，动用 Haskell 的这些强大的抽象能力仿佛有点杀鸡使用牛刀的感觉，不过一来这里只是为了展现 Haskell 的优点，二来对于大型程序来说，动作也是值的，这一性质无论说多重要都不为过。

此外，Haskell 还有许多强大之处文中并没有提到，如高阶函数，惰性求值，数据类型，多态，类型类（type class）等等，因为本文关注的是并发。Haskell 程序很少有像本文中的例子这样“命令式”的！如果想了解更多 Haskell 的知识，

可以访问 http://haskell.org，上面有书、指南、编译器、解释器、库、邮件列表和许多其他资源。

24.5 结论

本文的主要目的是要让你相信，用 STM Haskell 写出的并发程序从根本上比利用传统的锁和条件变量写出的并发程序的模块性更好。不过首先值得注意的一点是事务内存帮我们完全避免了基于锁的并发编程中种种令人头疼的经典问题（见“生锈的锁”一节）。所有这些问题在 STM Haskell 中完全消失不见了。Haskell 的类型系统会防止你在原子块之外读写一个 TVar，而且，由于不再存在对程序员可见的锁，因此加哪把锁、按什么顺序加锁的问题也就不复存在了。此外 STM 还有其他好处，但这里限于篇幅我写不下了（包括不再有忘记唤醒以及异常和错误恢复这些令人头疼的问题）。

不过，我最想说的一个问题还是可组合性（composability）问题，正如“生锈的锁”一节提到的，这是基于锁的编程中一个最严重的问题。但在 STM Haskell 中，任何 STM 类型的函数都可以顺序地或通过选择来与其他任何 STM 类型的函数组合起来形成一个新的 STM 型的函数，新的函数能确保具有组成它的各个函数的原子属性。特别是阻塞（retry）和选择（orElse）这两个功能，如果用锁来实现的话从根本上就不具备模块性，然而在 STM Haskell 中则是完全模块化的。例如，考虑下面这个事务，它用到的 limitedWithdraw 函数曾在“阻塞和选择”一节定义：

```
atomically (do { limitedWithdraw a1 10
               ; limitedWithdraw2 a2 a3 20 })
```

这个事务从阻塞中恢复的前提是账户 a1 上至少要有 10 块钱，而 a2 和 a3 之中则至少要有一个账户上多于 20 块钱。关键是，这么复杂的阻塞条件并不需要程序员显式写出，而且，如果 limitedWithdraw 系列函数是位于一个成熟的库中的话，程序员甚至根本不知道它们的阻塞条件是什么。总之一句话：STM 是模块性的，小的程序可以粘合成大的程序，无需暴露其内部实现。

本文只能说是对事务内存作了一个简单的概览，实际上事务内存还有许多其他有意思的主题文中没有提到，比如重要的有嵌套事务、异常、线程进展、饿死、不变式等。其中许多在关于 STM Haskell 的一些论文中都讨论过[23]。

[23] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy, "Composable memory transactions," ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05), June 2005; Tim Harris and Simon Peyton Jones, "Transactional memory with data invariants," First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06), Ottawa, June 2006, ACM; Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh, "Lock-free data structures using STMs in Haskell," Eighth International Symposium on Functional and Logic Programming (FLOPS '06), April 2006.

说实话，事务内存和 Haskell 可谓天造地设的一对。STM 的实现理论上可能要跟踪每一次内存读写，然而 Haskell 中的 STM 实现却只需跟踪 TVar 操作就行了，而 TVar 操作只占所有内存操作的极小一部分。此外，由于 Haskell 中的动作也是值，再加上 Haskell 丰富的类型系统，就使得我们无需对语言作任何扩展便能够提供强大的静态保证。不过话说回来，事务内存并非不适用于主流的命令式语言，虽然实现起来可能没这么优雅，并且可能需要更多的语言支持。目前 STM 是一个热门的研究课题；Larus 和 Rajwar 对这个领域的研究作了一个全面的概述[1]。

[1] James Larus and Ravi Rajwar, Transactional Memory, Morgan & Claypool, 2006.

STM 之于传统的并发编程技术就好比高阶语言之于汇编语言——你仍然还有可能写出有问题的程序，但许多棘手的 bug 却不可能再出现了，而且关注程序的高阶性质也使得编程容易得多。虽然并发编程并无银弹，但 STM 看起来是一个很有前途的进展，它能帮你编写出更美的代码。

24.6 致谢

很多人对本文的改进提了很好的建议和意见：Bo Adler, Justin Bailey, Matthew Brecknell, Paul Brown, Conal Elliot, Tony Finch, Kathleen Fisher, Greg Fitzgerald, Benjamin Franksen, Jeremy Gibbons, Tim Harris, Robert Helgesson, Dean Herington, David House, Brian Hulley, Dale Jordan, Marnix Klooster, Chris Kuklewicz, Evan Martin, Greg Meredith, Neil Mitchell, Jun Mukai, Michal Palka, Sebastian Sylvan, Johan Tibell, Aruthur van Leeuwen, Wim Vanderbauwhede, David Wakeling, Dan Wang, Peter Wasilko, Eric Willigers, Gaal Yahas, and Brian Zimmer。特别要感谢 Kirsten Chevalier, Andy Oram, 和 Greg Wilson 他们对文章作了非常详细的审阅。